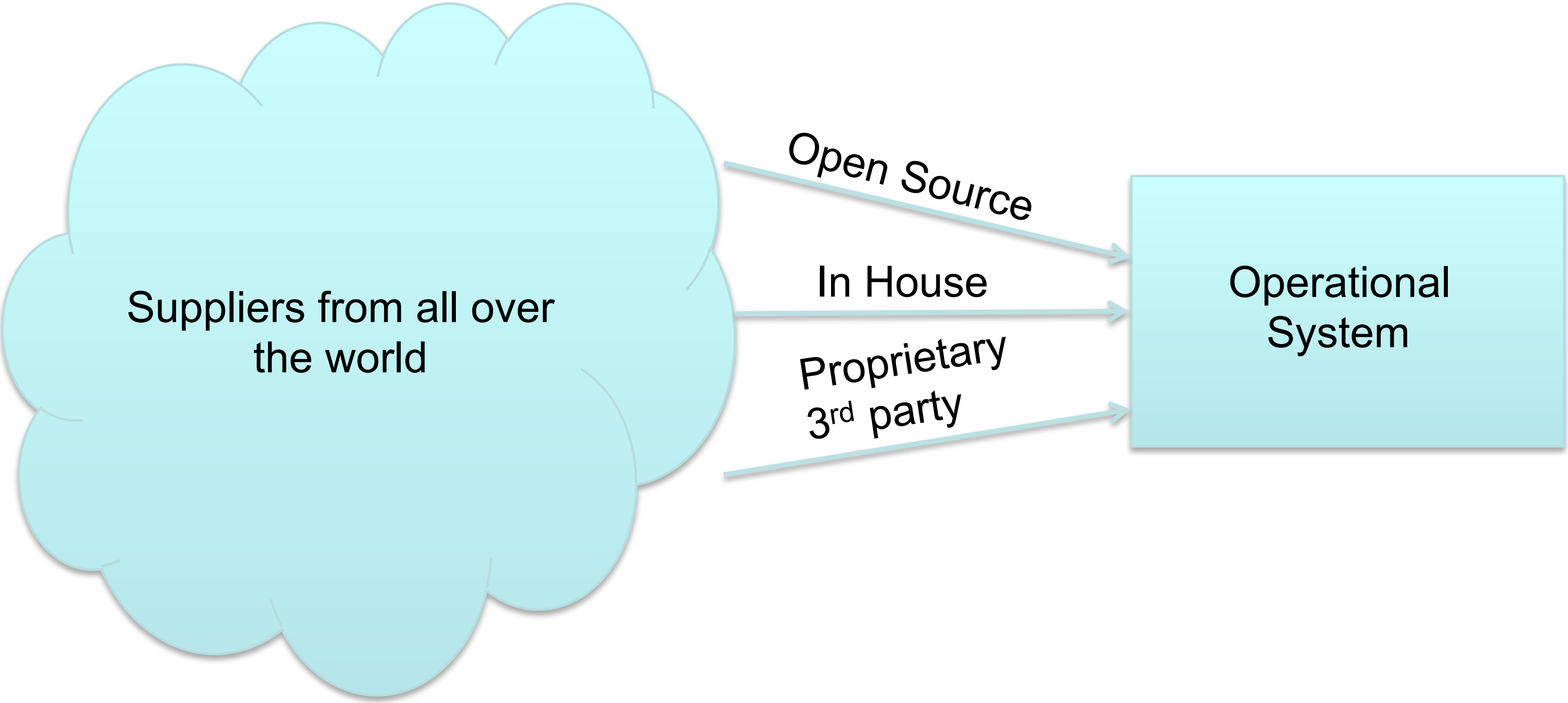
Securing the Continuous Deployment Pipeline

Len Bass, Ralph Holz, **Paul Rimba**, An Binh Tran, Liming Zhu

The software supply chain has a great deal of diversity



Suppliers from all over the world

Open Source

In House

Proprietary 3rd party

Operational System

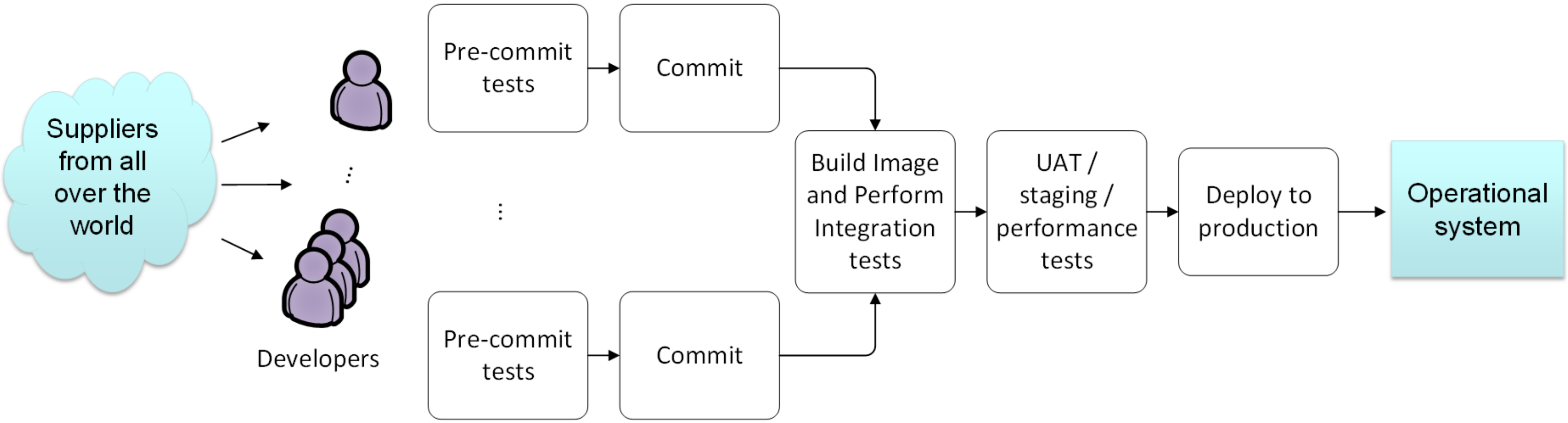
Many opportunities to corrupt delivery

- Rogue versions of 3rd party software
- Replace desired operational system with compromised version
- Leave “back door” in operational system
- Network access
- Credentials
- Software complexity

…

Deployment pipeline is the “last mile” of the supply chain

- The term “Last Mile” comes from telco and logistics
- It refers to the difficulties in getting goods and software to the consumer from a distribution centre



Suppliers from all over the world
Developers
Pre-commit tests
Commit
Pre-commit tests
Commit
Build Image and Perform Integration tests
UAT / staging / performance tests
Deploy to production
Operational system

Deployment pipeline

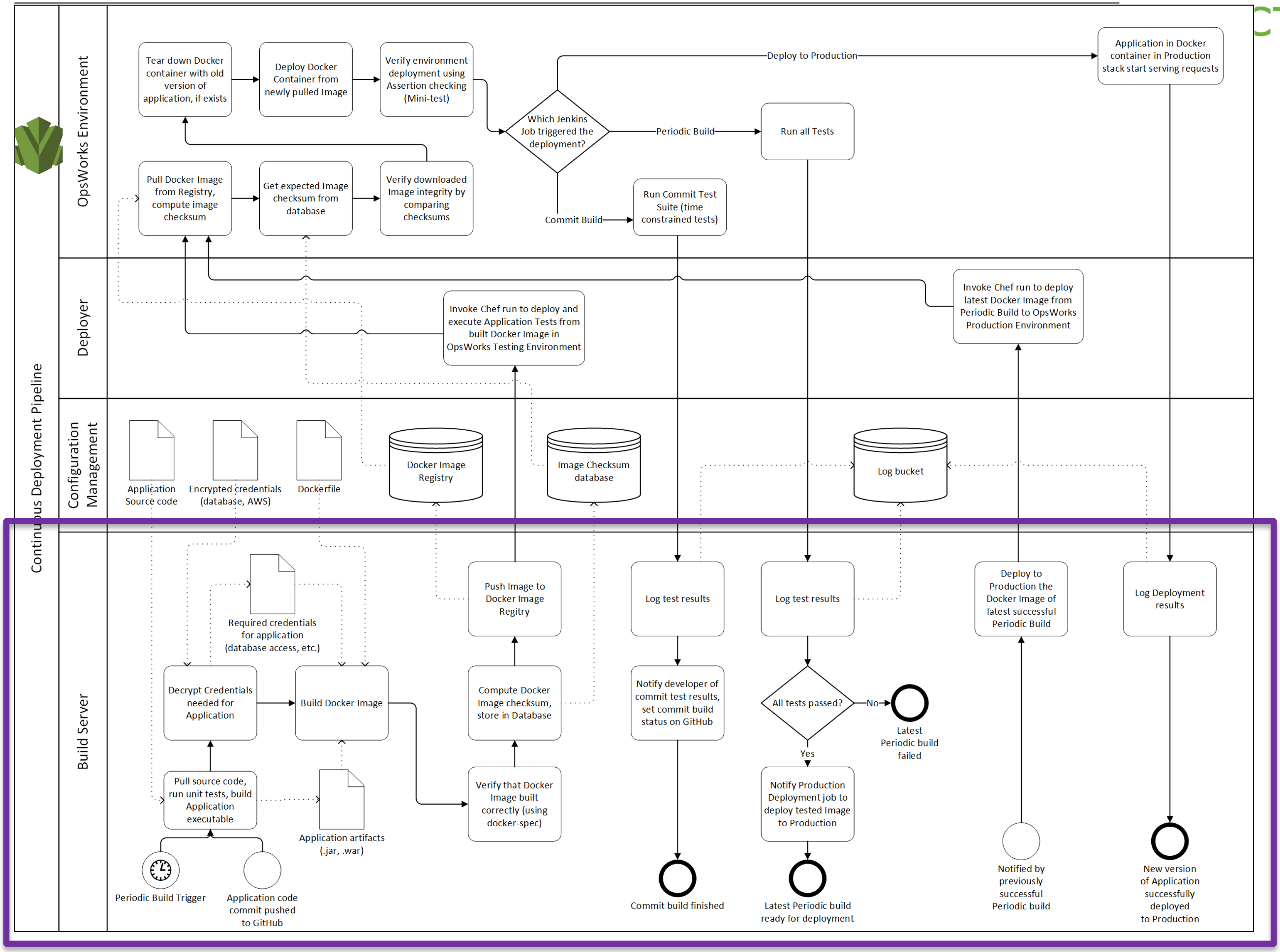
The security requirements and threats

- The security requirement we will discuss in this talk: the image deployed into operation is a valid image
  - This is an integrity requirement
    - The integrity of the specification of the image has not been compromised
      - Example violation: overwrite dockerfile
    - The image built is the image specified
      - Example violation: pulling the “wrong” version of code
    - The image deployed is the image built
      - Example violation: deploy wrong image
- Other security requirements exist but we do not focus on them in this talk

How do we secure a pipeline?

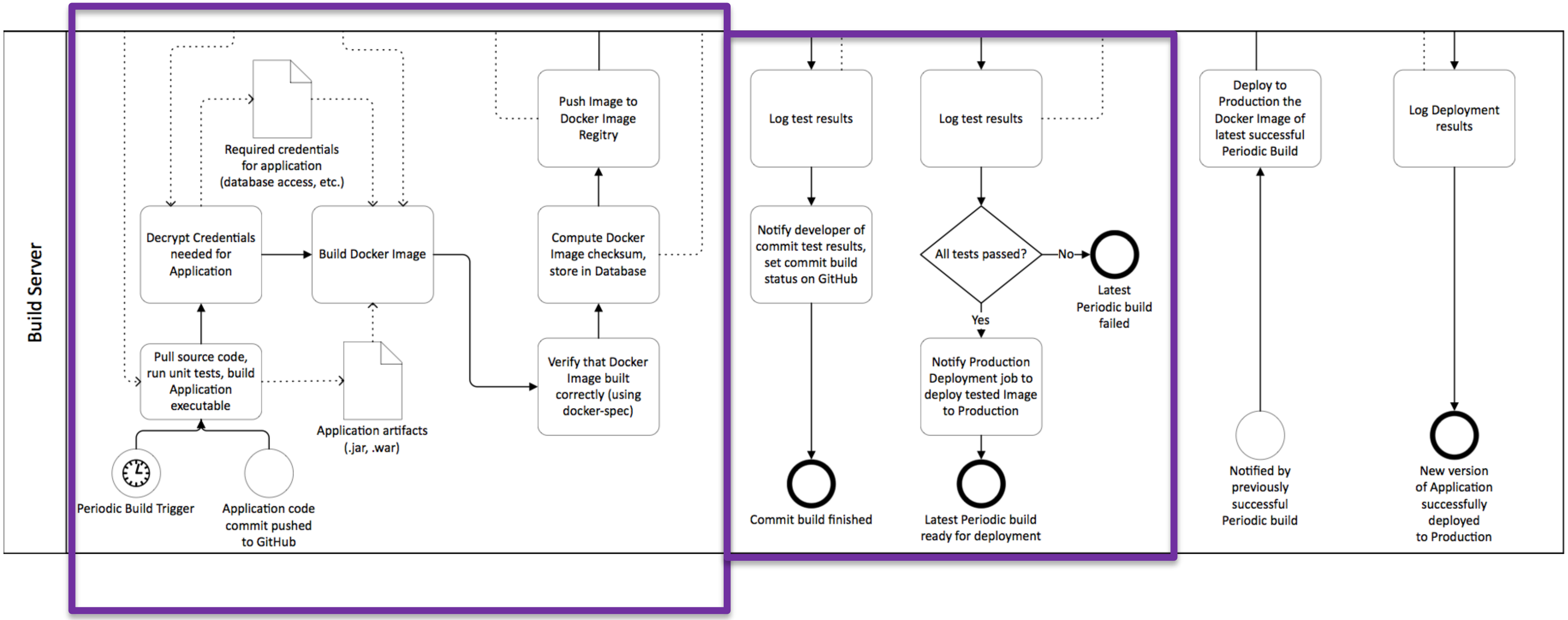
- Analyse a model of the pipeline to detect vulnerabilities (from design perspective)
- Restructure and remodel pipeline to remove vulnerabilities
- Ideally, we are able to remove all of the vulnerabilities. In this case the pipeline is "secure"
- Reality: we are not able to remove all vulnerabilities (at least not now). In this case, the pipeline has been "hardened"

A pipeline is complicated!!



Continuous Deployment Pipeline
OpsWorks Environment
Deployer
Configuration Management
Build Server
Tear down Docker container with old version of application, if exists
Deploy Docker Container from newly pulled Image
Verify environment deployment using Assertion checking (Mini-test)
Which Jenkins Job triggered the deployment?
Deploy to Production
Application in Docker container in Production stack start serving requests
Periodic Build
Run all Tests
Commit Build
Run Commit Test Suite (time constrained tests)
Pull Docker Image from Registry, compute image checksum
Get expected Image checksum from database
Verify downloaded Image integrity by comparing checksums
Invoke Chef run to deploy and execute Application Tests from built Docker Image in OpsWorks Testing Environment
Invoke Chef run to deploy latest Docker Image from Periodic Build to OpsWorks Production Environment
Application Source code
Encrypted credentials (database, AWS)
Dockerfile
Docker Image Registry
Image Checksum database
Log bucket
Required credentials for application (database access, etc.)
Push Image to Docker Image Regitry
Log test results
Log test results
Deploy to Production the Docker Image of latest successful Periodic Build
Log Deployment results
Decrypt Credentials needed for Application
Build Docker Image
Compute Docker Image checksum, store in Database
Notify developer of commit test results, set commit build status on GitHub
All tests passed?
No
Latest Periodic build failed
Yes
Pull source code, run unit tests, build Application executable
Application artifacts (.jar, .war)
Verify that Docker Image built correctly (using docker-spec)
Notify Production Deployment job to deploy tested Image to Production
Periodic Build Trigger
Application code commit pushed to GitHub
Commit build finished
Latest Periodic build ready for deployment
Notified by previously successful Periodic build
New version of Application successfully deployed to Production

A pipeline is complicated!!

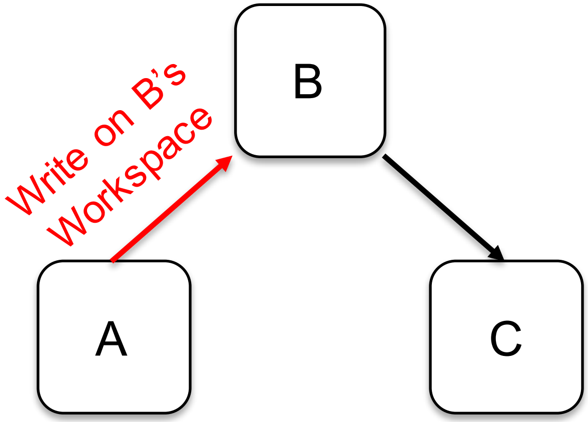


OUR PROCESS

Steps to harden the deployment pipeline

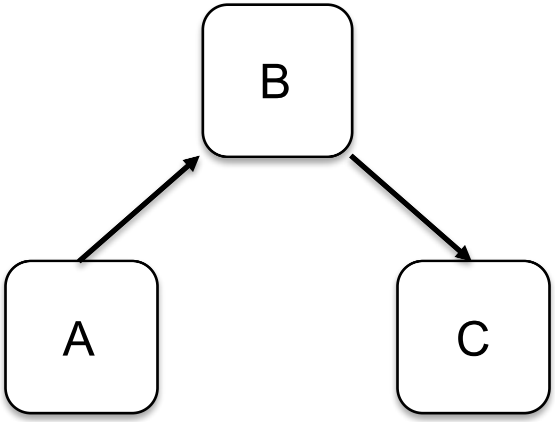
NICTA

- Identify security requirements to be satisfied
  - Apply principle of least privilege, isolation
    - No components should be able to damage other components
    - Communications between components are well specified and enforced



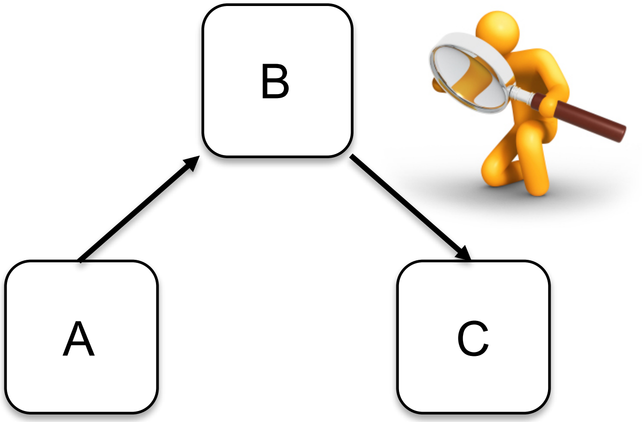
Steps to harden the deployment pipeline

- Repeat until all of the requirements have been satisfied OR can no longer decompose the untrustworthy components:
  - Model the interactions between the components
  - Analyse the model to check whether it satisfies our requirements
  - Decompose untrustworthy components causing an unsatisfied requirement into a trustworthy and an untrustworthy portion
    - Reduce the number of untrustworthy portions in the system
    - This is the "hardening" part



Steps to harden the deployment pipeline

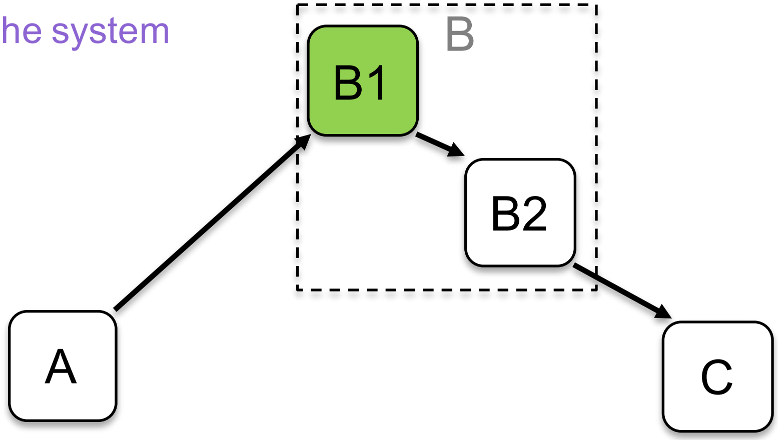
- Repeat until all of the requirements have been satisfied OR can no longer decompose the untrustworthy components:
  - Model the interactions between the components
  - Analyse the model to check whether it satisfies our requirements
  - Decompose untrustworthy components causing an unsatisfied requirement into a trustworthy and an untrustworthy portion
    - Reduce the number of untrustworthy portions in the system
    - This is the “hardening” part



Steps to harden the deployment pipeline

NICTA

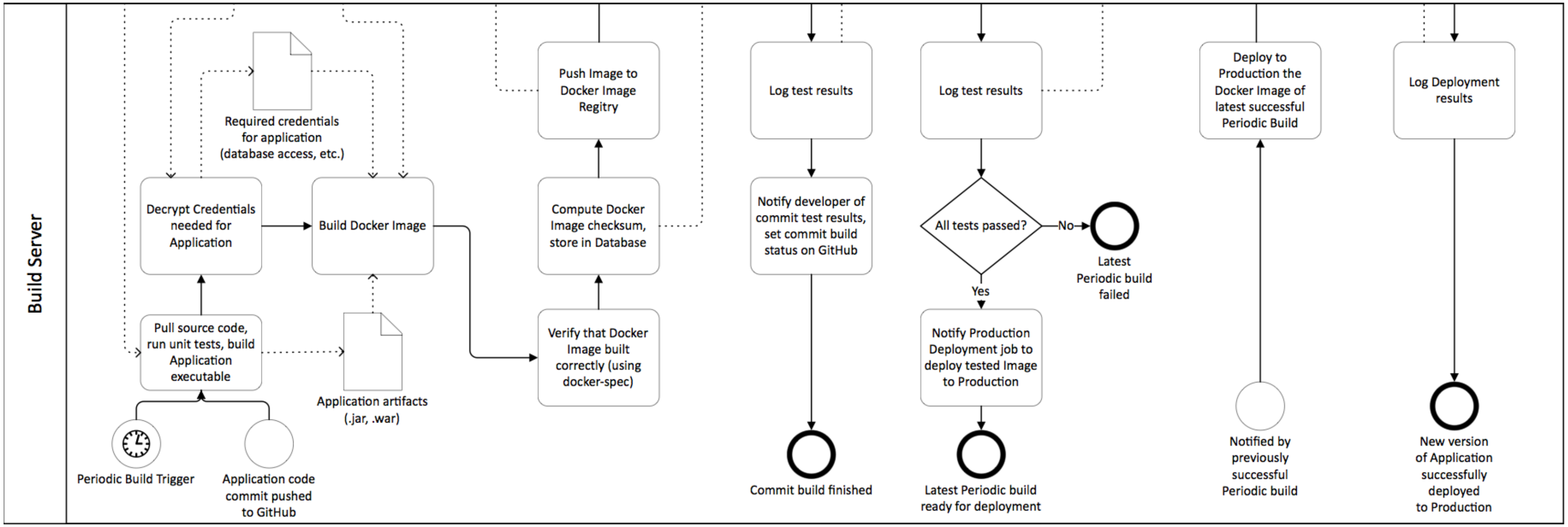
- Repeat until all of the requirements have been satisfied OR can no longer decompose the untrustworthy components:
  - Model the interactions between the components
  - Analyse the model to check whether it satisfies our requirements
  - Decompose untrustworthy components causing an unsatisfied requirement into a trustworthy and an untrustworthy portion
    - Reduce the LOC of untrustworthy portions in the system
    - This is the "hardening" part



- Implement new trustworthy components and modify untrustworthy components to utilize the trustworthy components to perform sensitive operations.

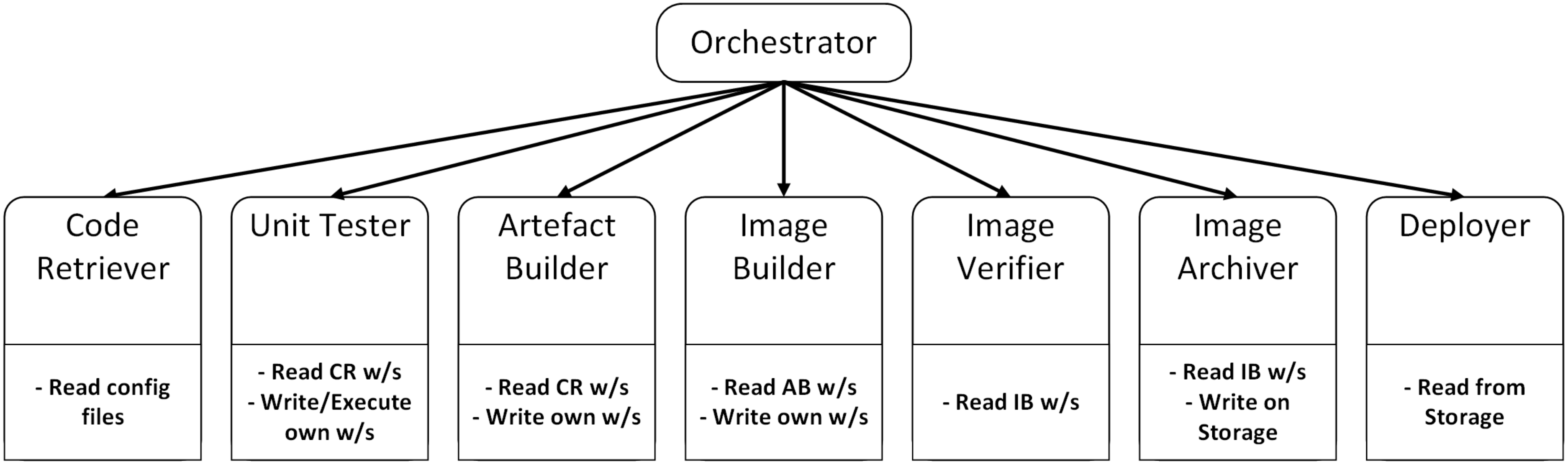
Original Build Server

- Build Server is a monolithic component
  - Large code-base
  - All the processes run under the same process space and privileges


Build Server
Required credentials for application (database access, etc.)
Decrypt Credentials needed for Application
Build Docker Image
Pull source code, run unit tests, build Application executable
Application artifacts (.jar, .war)
Periodic Build Trigger
Application code commit pushed to GitHub
Push Image to Docker Image Regitry
Compute Docker Image checksum, store in Database
Verify that Docker Image built correctly (using docker-spec)
Log test results
Notify developer of commit test results, set commit build status on GitHub
Commit build finished
Log test results
All tests passed?
No
Latest Periodic build failed
Yes
Notify Production Deployment job to deploy tested Image to Production
Latest Periodic build ready for deployment
Deploy to Production the Docker Image of latest successful Periodic Build
Notified by previously successful Periodic build
Log Deployment results
New version of Application successfully deployed to Production

Goal: Hardened Pipeline

- Orchestrator + Microservices
  - – Many microservices are small enough to be verified
    - We accept that not all can be verified
    - Verified for correctness (i.e. behave as specified)



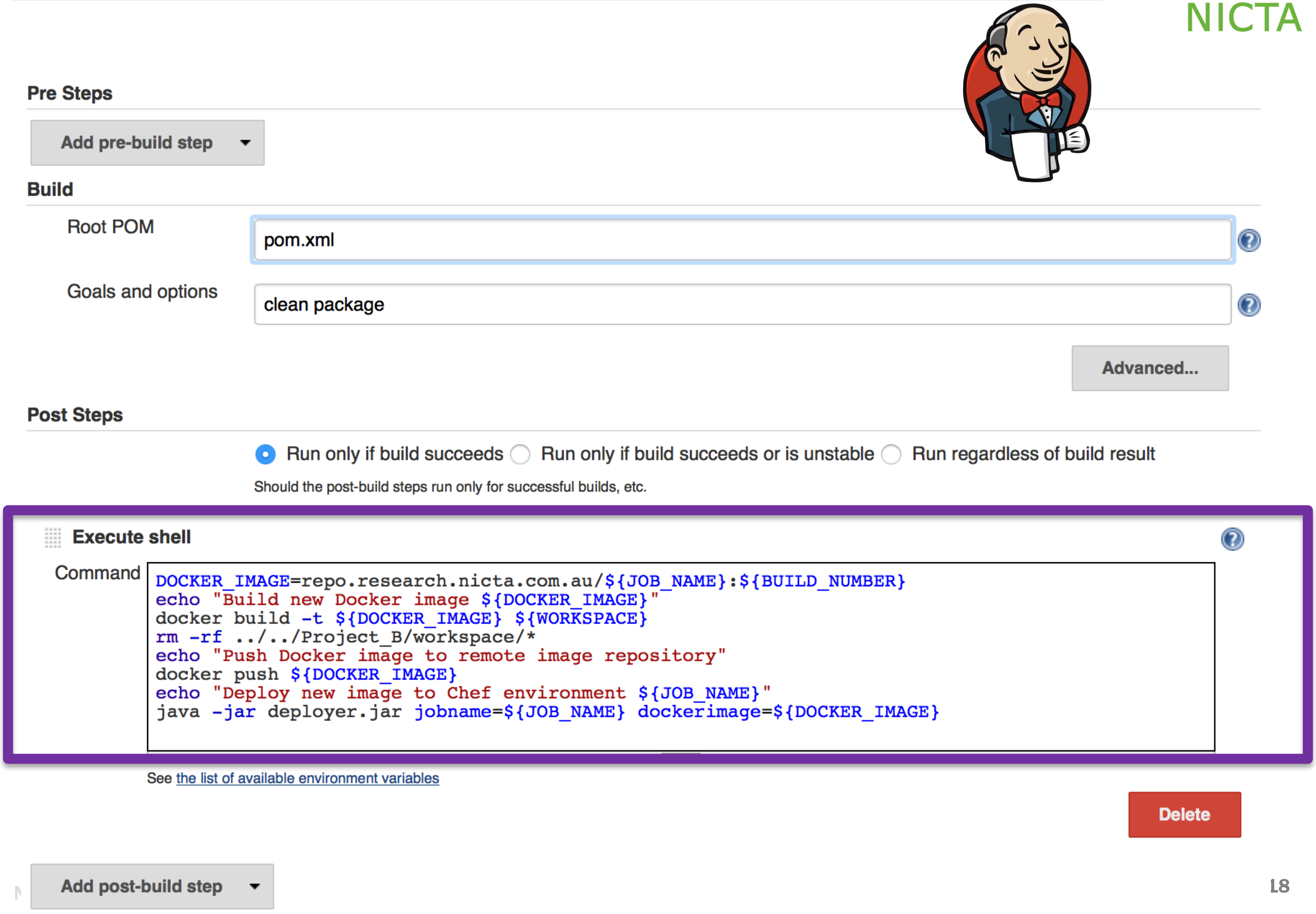
IN PRACTICE

From theory to practice

- We acknowledge reluctance to change
- Jenkins is the standard go-to build server
  - We use Jenkins as our build server
- Introduce a Jenkins plugin to enable microservices into the build server
  - Take advantage of Microservice architecture through well-defined API that we proposed
  - Microservices will do the actual work

Potential for damage

NICTA



Pre Steps

Add pre-build step ▾

Build

Root POM: pom.xml

Goals and options: clean package

Advanced...

Post Steps

● Run only if build succeeds ○ Run only if build succeeds or is unstable ○ Run regardless of build result

Should the post-build steps run only for successful builds, etc.

Execute shell

Command

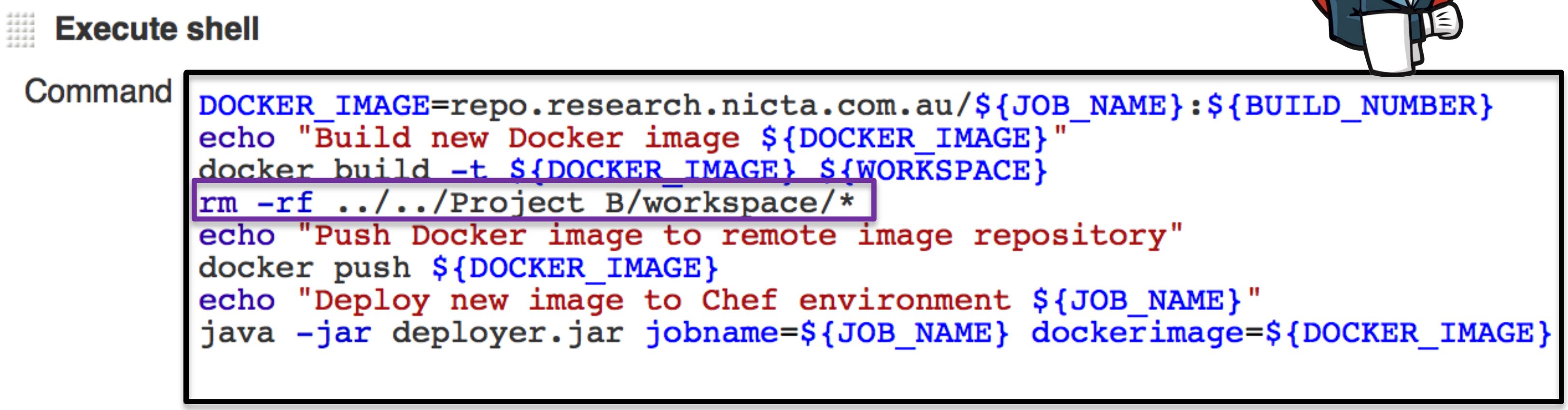
```
DOCKER_IMAGE=repo.research.nicta.com.au/${JOB_NAME}:${BUILD_NUMBER}
echo "Build new Docker image ${DOCKER_IMAGE}"
docker build -t ${DOCKER_IMAGE} ${WORKSPACE}
rm -rf ../../Project_B/workspace/*
echo "Push Docker image to remote image repository"
docker push ${DOCKER_IMAGE}
echo "Deploy new image to Chef environment ${JOB_NAME}"
java -jar deployer.jar jobname=${JOB_NAME} dockerimage=${DOCKER_IMAGE}
```

See the list of available environment variables

Delete

Add post-build step ▾

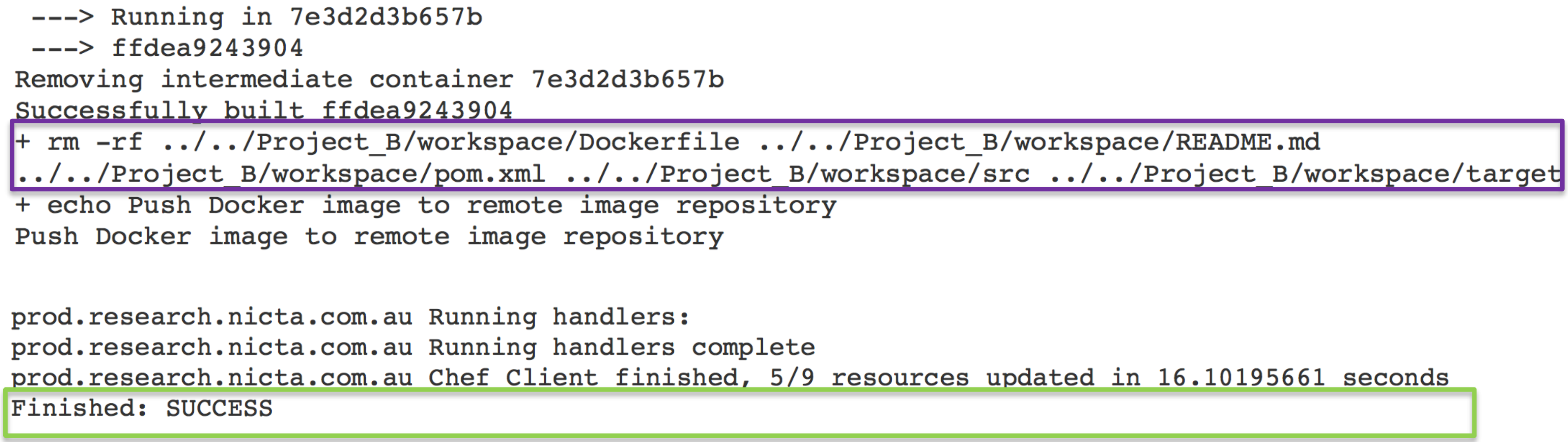
Potential for damage



Execute shell

Command

```
DOCKER_IMAGE=repo.research.nicta.com.au/${JOB_NAME}:${BUILD_NUMBER}
echo "Build new Docker image ${DOCKER_IMAGE}"
docker build -t ${DOCKER_IMAGE} ${WORKSPACE}
rm -rf ../../Project_B/workspace/*
echo "Push Docker image to remote image repository"
docker push ${DOCKER_IMAGE}
echo "Deploy new image to Chef environment ${JOB_NAME}"
java -jar deployer.jar jobname=${JOB_NAME} dockerimage=${DOCKER_IMAGE}
```

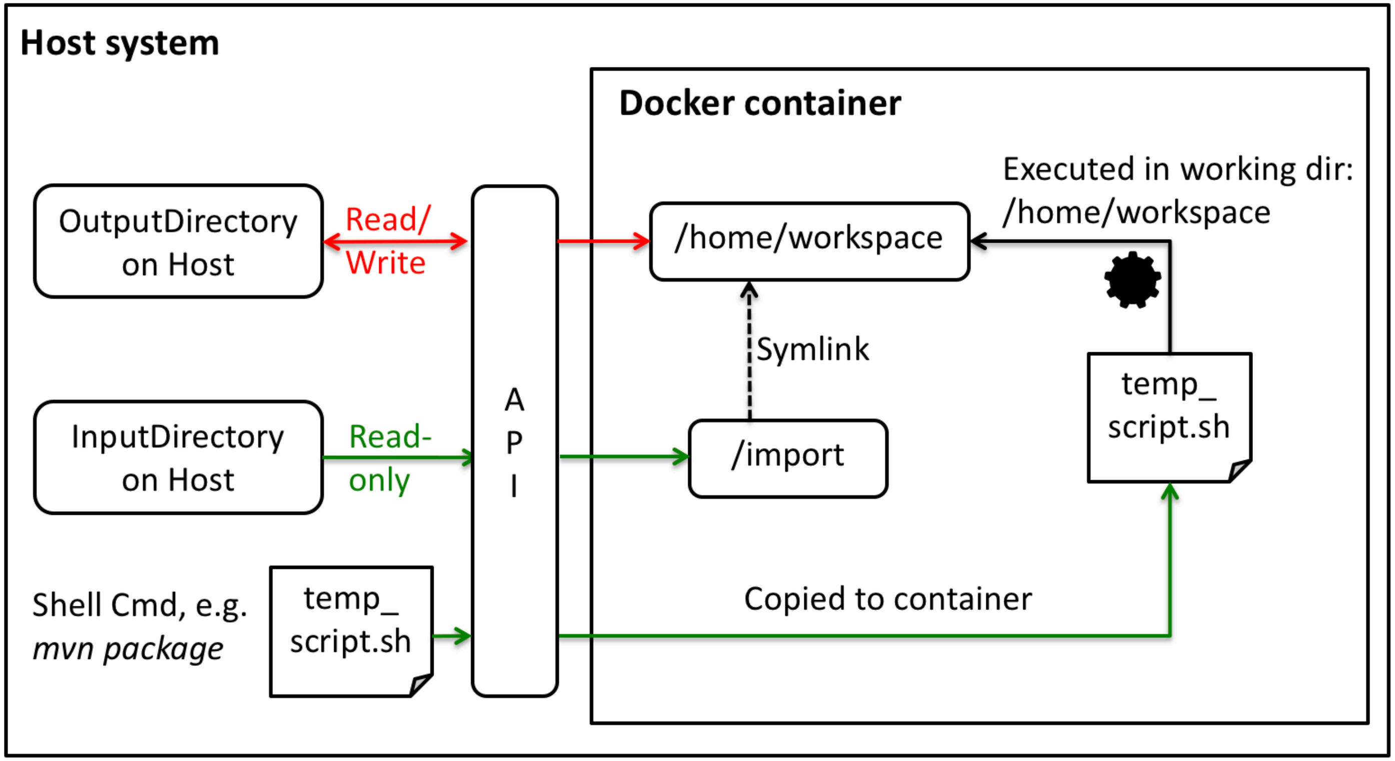


```
 ---> Running in 7e3d2d3b657b
 ---> ffdea9243904
Removing intermediate container 7e3d2d3b657b
Successfully built ffdea9243904
+ rm -rf ../../Project_B/workspace/Dockerfile ../../Project_B/workspace/README.md
../../Project_B/workspace/pom.xml ../../Project_B/workspace/src ../../Project_B/workspace/target
+ echo Push Docker image to remote image repository
Push Docker image to remote image repository


prod.research.nicta.com.au Running handlers:
prod.research.nicta.com.au Running handlers complete
prod.research.nicta.com.au Chef Client finished, 5/9 resources updated in 16.10195661 seconds
Finished: SUCCESS
```

One working solution: Sandbox shell

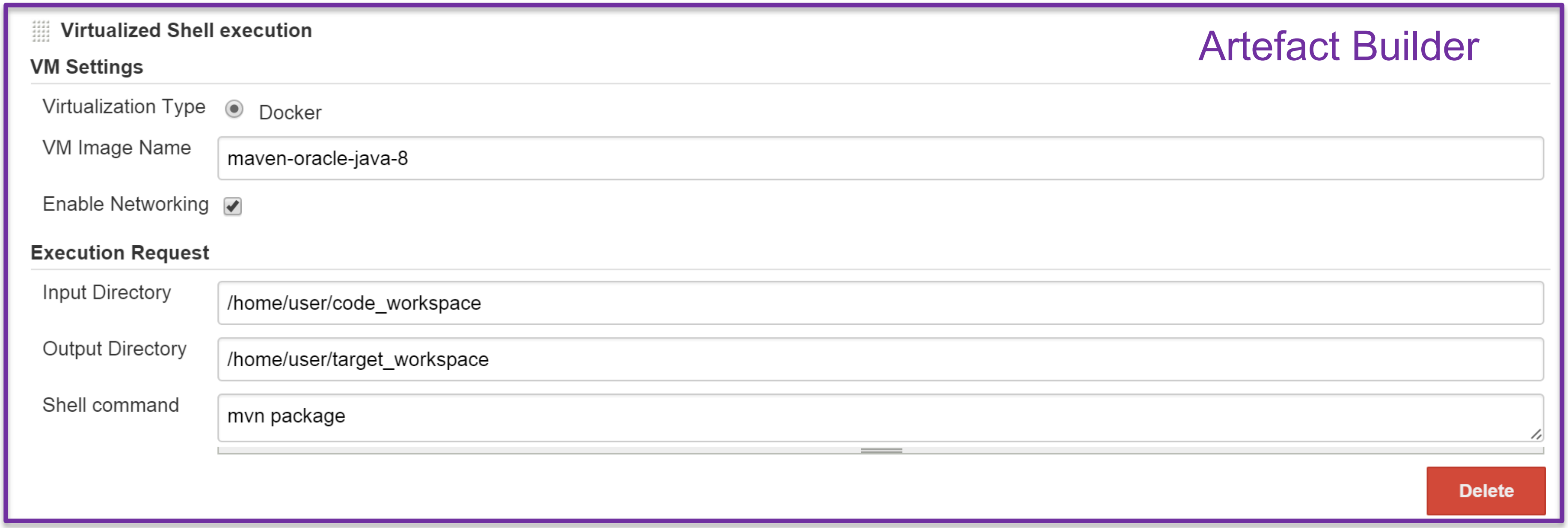
NICTA



- User only interact via API
- API functionalities
  - Spawn Docker container with specified VM settings (Image, CPU/RAM limit, etc.)
  - Map In dir (read-only) & Out dir (r/w access) to folders in container
  - Put shell commands into container
  - Security mechanisms enforcement

– Reduce attack surface on filesystem of Host to just the specified Out dir

Sandbox shell as Jenkins plugin



Artefact Builder

Virtualized Shell execution

VM Settings

Virtualization Type ● Docker

VM Image Name: maven-oracle-java-8

Enable Networking ☑

Execution Request

Input Directory: /home/user/code_workspace

Output Directory: /home/user/target_workspace

Shell command:

```
mvn package
```

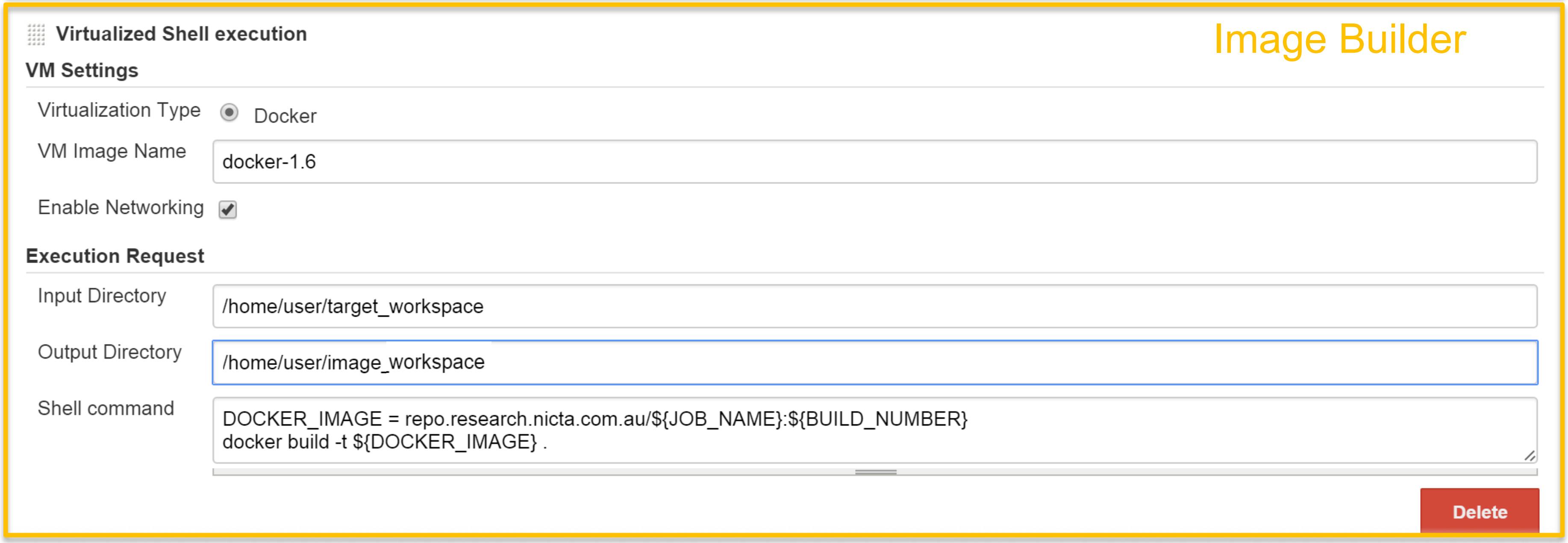


Image Builder

Virtualized Shell execution

VM Settings

Virtualization Type ● Docker

VM Image Name: docker-1.6

Enable Networking ☑

Execution Request

Input Directory: /home/user/target_workspace

Output Directory: /home/user/image_workspace

Shell command:

```
DOCKER_IMAGE = repo.research.nicta.com.au/${JOB_NAME}:${BUILD_NUMBER}
docker build -t ${DOCKER_IMAGE} .
```

Hardening the pipeline

- When we can fix some vulnerabilities but not all we say we have “hardened” the pipeline
- Our recommendations involve controlling access to resources (network, I/O, CPU, RAM)
- Ongoing: implementing micro components that communicate with Jenkins
- Ongoing: formal verification on the micro components

Summary

- Our contributions are
  - The creation of an engineering process to evaluate/modify the design of a deployment pipeline
  - Architectural recommendations for the tools in the pipeline
  - Presented one practical example of hardening a pipeline
    - A plugin that enables microservice architecture
    - Sandbox shell
- Our process is based on
  - Identifying trustworthy components,
  - Patching vulnerabilities by creating small trustworthy components,
  - Refining until no vulnerabilities remain.
- The specifics of what we have done depends on the technologies we use but the process will work for any collection of technologies.